A Case Study in Preserving a High Energy Physics Application

Haiyan Meng, Matthias Wolf, Peter Ivie, Anna Woodard, Michael Hildreth, and Douglas Thain
Department of Physics and Department of Computer Science and Engineering
University of Notre Dame
{hmeng|mwolf3|pivie|awoodard|mhildret|dthain}@nd.edu

ABSTRACT

The reproducibility of scientific results increasingly depends upon the preservation of computational artifacts. Although preserving a computation to be used later sounds easy, it is surprisingly difficult due to the complexity of existing software and systems. Implicit dependencies, networked resources, and shifting compatibility all conspire to break applications that appear to work well. To investigate these issues, we present a case study of a complex high energy physics application. We analyze the application and attempt several methods at extracting its dependencies for the purposes of preservation. We propose one fine-grained dependency management toolkit to preserve the application and demonstrate its correctness in two different environments - one virtual machine from Notre Dame Cloud Platform and one virtual machine from Amazon EC2 Platform. We report on the completeness, performance, and efficiency of each technique, and offer some guidance for future work in application preservation.

1. INTRODUCTION

Reproducibility is a cornerstone of the scientific process [5]. In order to understand, verify, and build upon previous work, one must be able to first recreate previous results by applying the same methods. Historically, reproducibility has been accomplished through painstaking detailed documentation recorded in lab notebooks, which are then summarized in peer-reviewed publications. But as science increasingly depends on computation, reproducibility must also encompass the environments, data, and software involved in each result [34]. It is widely recognized that informal descriptions of software and systems – although common – are insufficient for reproducing a computational result accurately. A more automated and comprehensive approach is required.

The reproduction of a computation has three broad components, each of which suggests somewhat different approaches:

- The **computing environment**, consisting of the basic hardware and the operating system can be preserved as physical artifacts or as a combination of virtual machine monitor (hardware) and virtual machine image (operating system) [24].
- The **scientific data** to be analyzed has historically received the most attention for curation. In a large, well-organized project, it may be stored in a data repository or database management system, with associated documentation and a curation strategy. In a small effort, it could simply be a handful of files.
- The **software environment** includes the source code, binaries, scripts, configuration files, and everything else needed to execute the desired code. As with data, the software could be drawn from a well-managed software repository, or it could be a handful custom scripts that exist in the user's home directory.

In a very abstract sense, reproducing a computation is trivial. Assuming a computation is deterministic, one must simply preserve all of the inputs to a computation, then re-run the same code in an equivalent environment, and the same result will be produced. For a small custom application on a modest amount of data, this could be accomplished by capturing the complete environment, data, and software within a single virtual machine image, and then depositing the virtual it into a curated environment. The publication could then simply refer to the identifier of the image, which the interested reader can obtain and re-use. This approach has been used to some success with systems mentioned in [8]. [1]

However, this simple approach is not sufficient for large applications that are run in complex social environments.

- There may be **implicit dependencies** on items that are not apparent to the end user. For example, they may understand that they rely on a particular data analysis package, but would have no reason to know that the package has further dependencies on other libraries and configuration files. Or, they may know that the computation only runs correctly on a particular machine, but not know this is because it relies on a filesystem that is mounted only on that machine.

[1] Of course, we are glossing over the problem that hardware architectures and virtual machines also change, so one must also preserve the VMM software necessary to run the image. The VMM itself depends on a software environment which must also be preserved. A long-term preservation system might end up running a whole stack of nested virtual machines in order to provide the desired environment!

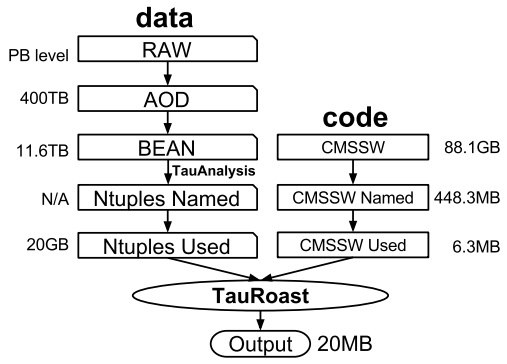

Figure 1: Inputs to Tau Roast

- The **granularity** of the dependencies may not be well understood. For example, the user may understand that a computation depends upon a data collection that is 1TB in overall size, but not have detailed knowledge that it only requires three files totalling 300MB out of that whole collection.

- There may be dependencies upon **networked resources** that are inherently external to the system, such as a database, a code repository [11], or a scalable filesystem [4]. For such resources, it must be decided whether the dependency will simply be noted, or if it must be incorporated whole or in part.

- Where **common dependencies** are widely used, it may be inefficient or impossible to store one copy of each dependency for each archived object. Some form of sharing or de-duplication is necessary in order to keep the archive to a reasonable size.

We do not claim to have solved these problems in any comprehensive way. Rather, our aim in this paper is to highlight the scope of the problems by presenting a case study of one complex application. The application is presented to us first in the form of an email that describes in prose how to install the software and run the analysis. We perform several successive refinements to convert it into an executable and preservable object. We then develop techniques for reducing the size of the dependencies that are necessary for the object to function, and we demonstrate the preserved object functioning correctly in three different physical and cloud environments. We describe how each of these techniques may interact with a future archive of preserved software artifacts, and conclude with some reflections on the challenges of preservation and advice for future efforts.

2. OVERVIEW OF TAU ROAST

Within the ongoing investigation of the Higgs boson at the CMS detector, part of the LHC at CERN [10], the Higgs production in association with two top quarks allows to measure the Higgs coupling strength to top quarks. As the Higgs boson is to short-lived to be detected itself, it has to be reconstructed from its decay products.

The application which is the study of this paper is called *TauRoast*. It searches for cases where the Higgs boson decays to two tau leptons. The leptons are not observed directly, but by the particle showers that they generate. So, the analysis must search for detector events that show a signature of decay products compatible with both hadronic tau and top decays. Properties of such events are used to distinguish the events of interest (Higgs decays) from all other events and are also used in further statistical analysis.

Name	Location	Total	Named	Used
CMSSW code	CVS	88.1GB	448.3MB	6.3MB
Tau source	Git	73.7MB	73.7MB	6.7MB
PyYAML binaries	HTTP	52MB	52MB	0KB
.h file	HTTP	41KB	41KB	0KB
Ntuples data	HDFS	11.6TB	N/A	20GB
Configuration	CVMFS	7.4GB	N/A	103MB
Linux commands	localFS	110GB	N/A	68.4MB
HOME dir	AFS	12GB	N/A	32MB
Misc commands	PanFS	155TB	N/A	1.6MB
Total		166.8TB	N/A	21GB

The first column illustrates the total size of each data and software source; the second column illustrates the size of the named files from each source; the third column illustrates the size of actually used data from each source. N/A denotes it is hard to figure out the named size of implicit dependencies directly.

Table 1: Data and Code Used by Tau Roast

Figure 1 shows that both the code and data that form *TauRoast* are drawn from large repositories through multiple steps of reduction. A preservation strategy must weigh whether to store the large repositories completely, the fragments used by an artifact, or something in between.

Code Sources. Like many scientific codes, the central algorithm of *TauRoast* is expressed in a relatively small amount of custom code developed by the primary author. But, the code cannot run at all without making use of an enormous collection of software dependencies. Some of these dependencies are standard to operating systems worldwide, some are standardized across the entire high-energy physics field, some are particular to small collaborative groups, and a few are very specific to a single researcher.

The largest of these repositories is the CMS Software Distribution (CMSSW), a carefully-curated selection of software packages which is distributed in several forms. Historically, components of CMSSW were obtained by checking components of the source out of CVS, or by installing a complete binary package on a shared filesystem within an HPC center. In recent years, distribution has moved to an on-demand delivery system known as CVMFS [4], which provides a filesystem interface that transparently accesses a remote repository. The content of CMSSW is managed very carefully by a centralized team whose main goal is to ensure that the current version of the software operates correctly on the operating systems and architectures currently in use. However, preservation is not a specific objective of the system, and so there is no particular guarantee that old versions of CMSSW will continue to operate indefinitely.

CMSSW contains many different tools, libraries, and utilities. No single code uses anywhere close to all of that. But, because it is widely used within the community, it is common for users and developers to simply expect that a particular version of the entire repository is available.

Data Sources. The CMS collaboration provides analysis end-users with a pre-processed and reduced data format, AOD [15], containing information for events, i.e., proton-proton collisions with a signature of interest, in the form of

reconstructed particles. This format is based on the RAW output of the CMS detector readout electronics and reconstructed world-wide. Both real and simulated data are available for examination.

As AOD data are too large to be iteratively processed repetitively in an physics analysis workflow, it is normally reduced further in structural complexity and content. For the analysis under investigation here, this is a two-step process. First, the AOD data are processed at the Notre Dame working group cluster to BEAN events, containing only trivial data containers packed in vectors. This step is time and CPU intensive and its output contains data of 11.6 TB to be analyzed by the tau analysis. It is performed by a small custom code framework, which is built on top of CMSSW. The BEAN format, production code, and data are shared within the analysis group looking at Higgs production in association with top quarks, which is formed by groups from a few American and European universities, consisting of up to a few dozen contributors.

In the second step, the data are reduced to the "Ntuple" format, which contains only events matching basic quality criteria and fields relevant to *TauRoast*. This results in a dataset of 43.3 GB. Again, the Notre Dame CMS groups cluster resources are used to perform this reduction and selection, running highly customized software, built on CMSSW and the BEAN framework, with code written and maintained by a small group.

Once the data has been reduced to Ntuples, TauRoast can be run as a single process, and contains a stringent event selection to keep only high quality candidate events for the underlying physical process (using about 20 MB of space). Quantities from the selected events can be both plotted and used in multivariate analysis to determine the level of expected signal in real data. This package is written using the CMSSW build framework, but only utilizes code from ROOT, a particle physics toolkit underlying CMSSW, and a few external python dependencies for convenience.

3. OBSERVATIONS

TauRoast was provided to us in the form of an email which described, in prose, how to obtain the source, build the program, and run it correctly on one specific machine at our home institution, with no particular guarantee that it will run anywhere else in the world. Although this starting point may seem extreme, it is perfectly natural for collaborators to share configurations with each other in this form, and to rely on the presence of a working environment with appropriate dependencies already installed. From this starting point, the authors played the role of curators, whose job it is to prepare the application for permanent archival.

First, we elaborated the email instructions into an executable script that obtains the dependencies and then executes the analysis. The script declares the necessary environment variables, downloads and checks out the necessary source code, builds it appropriately, calls initialization scripts in the dependent software, and then runs the analysis. A few rounds of correction with the original author were necessary to obtain all the dependencies and run the artifact correctly. (The original email also indicated how to run the application within a production batch system. For the purposes of preservation, we consider the execution infrastructure to be distinct from the application, and leave it out of consideration for now.)

The process of elaborating the program into a script revealed several observations about this type of application:

- **Many Explicit External Dependencies.** TauRoast depends on a large number of external dependencies, each with a different access method and data source. While we knew in advance that it depended upon the large CMSSW distribution, it was not apparent until elaborating the script that it depended upon two different Github repositories for the Tau source, a CVS server at CERN for some configuration information, a public web page for the PyYAML library, and the public home page of a Notre Dame student for one missing header file. (The latter is particularly troubling!) While, at some level, the authors and users of these software know of these dependencies, they are often missing in informal communications or forgotten once the dependency is installed. However, once known, they are at least expressed explicitly within the script.

- **Many Implicit Local Dependencies.** A much harder problem is that the application assumed the presence of many different components in the local filesystem view. It would be tempting to capture all of these by simply storing a virtual machine image containing the local filesystem. However, the application depended on no less than **five** networked filesystems available on a particular machine available to the author: the data to be analyzed was stored on a HDFS [6] cluster, some configuration data was stored on a CVMFS [4] filesystem, and a variety of software tools were on a NFS [17], PanFS [33] and AFS [28] systems. The original authors were not aware of many of these dependencies, because they simply relied on local administrators to configure the software and make it available.

- **Configuration Complexity.** As a means of controlling the complexity of dependent software packages, the high energy physics community has developed a number of tools that perform run-time configuration and consistency checks of the available software. `scram` is the software management tool used by the CMS experiment. Before running any code, `scram` is used to locate the appropriate version software, set environment variables such as the PATH, run any tool-specific configuration, and do the same for all software on which it depends. If the correct versions are not available, `scram` halts and emits an error. While this procedure has great value for consistency, it also introduces a significant cost because it involves a large number of nested scripts traversing a filesystem, repeatedly looking up metadata. In our example, the time to perform this configuration with a cold cache is about 14 minutes, which is almost as large as the actual analysis run, which takes 20 minutes.

- **High Selectivity.** Although the total size of the resources accessed by this program is very large, the size of the data and software actually used are much smaller. Often, an entire repository or data source is named within the script, but the program only needs a handful of items from that source. For example, the data is stored on an HDFS filesystem with 11.6TB of

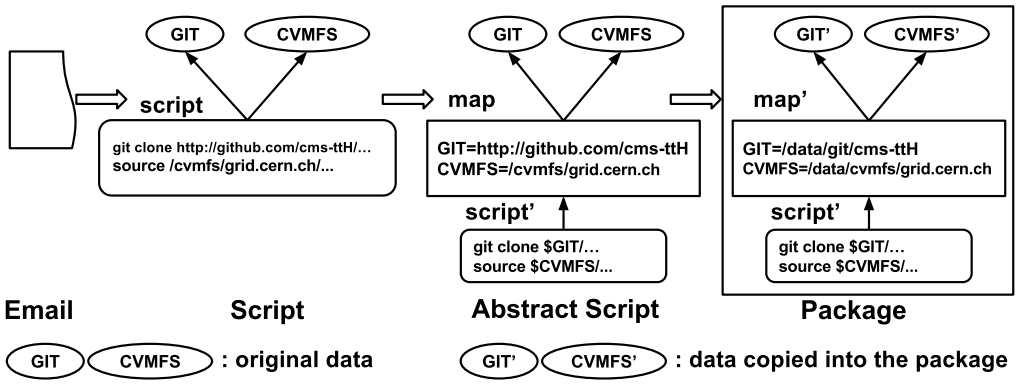

Figure 2: Version Evolution

data, but only 20GB are actually consumed by the program. The CMSSW repository is 88.1GB in total but only 448.3MB in source are checked out, and the software actually used only measured 6.3MB. In a few cases, a source of software is named but never actually accessed. (For example, our original script includes the Open Science Grid software stack in the PATH, but does not actually use it.) We suspect that end users are accustomed to missing dependencies and thus get in the habit of adding commonly used software, whether it is needed or not.

- **Rapid Changes in Dependencies.** Over the course of three months between collecting the initial email, analyzing the program, and writing this paper, the computing environment was under continuous change. The CMSSW software distribution released a new version, the target execution environment was upgraded to a new operating system, and the application deprecated the use of CVS for obtaining the software. While the users of this software seem be accustomed to constant change, any preservation technique will have to be very cautious about relying upon an external service, even one that may appear to be highly stable.

4. EVOLVING THE ARTIFACT

It is clear that the artifact, as provided, is not in a suitable form for preservation. While it might be technically possible to automatically capture the entire virtual machine and all of the connected filesystems, it would require 166.8TB of storage, which would be prohibitively expensive for capturing this one application is alone. Further, if multiple similar applications are preserved, we would miss the opportunity to identify common dependencies and store them once for multiple artifacts. A more structured approach to dependency management is needed.

Figure 2 shows how we have evolved this artifact through several stages which make it more suitable for preservation. In each step of evolution, we make the dependencies of the artifact more explicit and available for analysis and automated processing. As noted in the previous section, the original author provided us with prose instructions by email which we translated into an executable script. The executable script has embedded in it a number of external identifiers such as URLs pointing to repositories and paths to networked filesystems. As a general programming practice, embedding such constants into the middle of a program is unwise, and so we extract all of those identifiers and place them *outside* the script in a *dependency map* or just *map* for short. The dependency map lists all of the external dependencies of the application, indicating the type, how they are accessed, and where they are currently located. The resulting script then simply refers to abstract file locations such as `GIT` and `CVMFS`, while the map file indicates where they are currently located. If properly constructed, the script should not refer to any external resource unless it is indicated in the dependency map. We call this idealized artifact an *abstract script*.

By extracting the dependencies into the dependency map, we introduce great freedom for the curator to move, transform, and otherwise manipulate the dependencies of the artifact without damaging the artifact itself. A Figure 2 shows, it is straightforward for an automated tool to examine all of the dependencies in the map, download those that are missing, and then modify the map to point to the local copies of the dependencies. If we group the script, dependency map, and dependencies into a *package*, we now have a self-contained artifact that can be moved from place to place. In some cases, it may be safe to allow the dependency map to refer to trusted remote repositories. Whether this is advisable is a judgment that must be made by the user or the curator, taking into account the long-term stability of said repositories.

However, only having the package and the dependency map is not enough. The successful execution of the application on the original machine also relies on the configuration of environment variables. We collected the environment variables of the original machine and transformed it into one executable script. If another researcher wants to repeat the application, this executable script will be first executed.

The relationship of different roles involved in the application preservation and reproduction is shown in Figure 3. The original author uses the packaging utility to generate the package for one application. Then the package, together with its map file and description file will be published. When another researcher wants to repeat the application, one copy of the package will be downloaded into the new machine and the application can be repeated.

When we try to repeat one application on one new machine, one map file is necessary for the relocation of the data

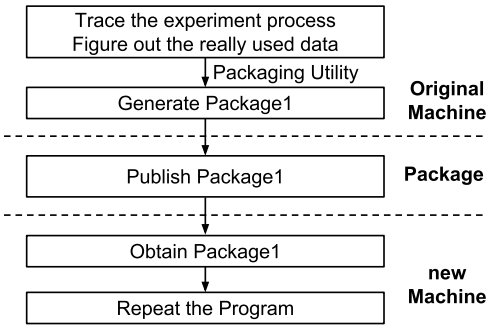

Figure 3: Relationship of Roles

access targets, as show in Figure 2. The map file clearly defines the real location of each dependency in the format of dependency variables - the real location of dependency variable `GIT` is `/data/git/cms-ttH` and the real location of dependency variable `CVMFS` is `/data/cvmfs/grid.cern.ch`. The script only refers to the dependency variables defined in the map file. This design decouples the application script and the actual data access targets, which minimizes the impact of the evolution of different data dependencies and ensures the transparent access. The modification of the package only introduces the minimal changes of the map file on the client side.

This basic approach to dependency management is a step in the right direction for dependencies that are *explicit* and *external* to the user's native execution environment. However, it leaves two other problems unsolved.

First, the basic approach requires that someone be *aware* of the dependencies, whether it be the end user, the system administrator, or the archive curator. It seems reasonable to expect the user to be aware of a large dependency mentioned in a top-level script. But, oftentimes the dependency is embedded invisibly deep within the software stack, or is connected to the machine by the local system administrators. No single party is likely to have complete information about all of the dependencies. Second, the basic approach assumes that the entire dependency is actually consumed by the artifact. As we have suggested above (and will show below), this sort of program often only consumes a small fraction of what it does declare as a dependency.

To address both of these problems, users and curators alike need tools that will automatically observe the dependencies of complex applications, to facilitate automatic and efficient preservation.

5. MEASURING DEPENDENCIES

We have developed a prototype tool to assist in the measurement and preservation of implicit dependencies for complex applications. We use Parrot [32] to explicitly record all of the files accessed by our example application, allowing us to observe how much of each external dependencies is used, and what local resources are implicitly used. Using this information, we create a *reduced package* which contains only the files actually used by the application.

Parrot is a virtual filesystem access tool which has previously been used to attach existing programs to a variety of remote I/O systems, such as HTTP, FTP, and CVMFS. It works by trapping an application's system calls through the `ptrace` debugging interface, and then replacing them with the desired I/O operations. Parrot is already used in the high energy physics community with applications like TauRoast specifically to provide access to the CMSSW software distribution via the CVMFS distributed file system. We made small modifications to Parrot to record a *namelist* which lists all of the files that an application actually accesses.

Path used in Program	Actual Location
`/`	`/tmp/package-hep`
`/tmp/package-hep`	`/tmp/package-hep`
`/dev`	`/dev`
`...`	`...`

Table 2: Structure of Map File

Figure 4 illustrates the measurement process. The starting point of this toolkit is one successful execution of the application on the native machine. First, we execute the actual data analysis script under Parrot to generate the namelist. Then, using the namelist, we generate a package containing all the necessary data and software for one analysis program is generated. When another researcher wants to repeat the program, he only needs to obtain the package and execute the actual analysis program inside the package.

For one execution of *TauRoast*, the namelist 132,047 accessed filenames, along with the operation used to access the file, such as `open`, `stat`, `read`, etc. With duplicate filenames removed, the list is reduced to 67,168 files. Many of those entries do not exist, because they reflect attempts by the application to search for programs and libraries in multiple places. Only 22,068 entries reflect existing files or directories.

The packaging tool iterates over each item of the filename list, determines the process mode and replication degree according to the file type (common files, directories, symbolic links) and the system call type, generates one package containing the dependencies, and summarizes the contents of the package as shown in Table 3. To the extent possible, the filesystem structure of the original environment is preserved.

We considered several approaches to constructing the package. In a **shallow copy**, we only copied the individual files in the namelist, creating only parent directories for each. Where a directory was listed, we created the directory and populated it with empty files as placeholders to facilitate a directory listing. In a **medium copy**, we copied the individual files as before. Where a directory was listed, we created the directories and copied the contents of the files in that directory, one level deep. A **deep copy** would duplicate all directories recursively, but this would have resulted in TB-sized packages, so we did not consider it further.

Parrot is required to re-run the packaged artifact, in order to force the packaged files to appear to exist in their original locations. To this end, the packaging tool creates a *file map* which maps the logical names of the files to their current physical locations, as shown in Table 2. Parrot reads the file map and redirects system calls at run-time to achieve the desired effect. As the example suggests, special device files such as `/proc` and `/dev` are not incorporated into the package but are instead accessed natively.

6. EVALUATION

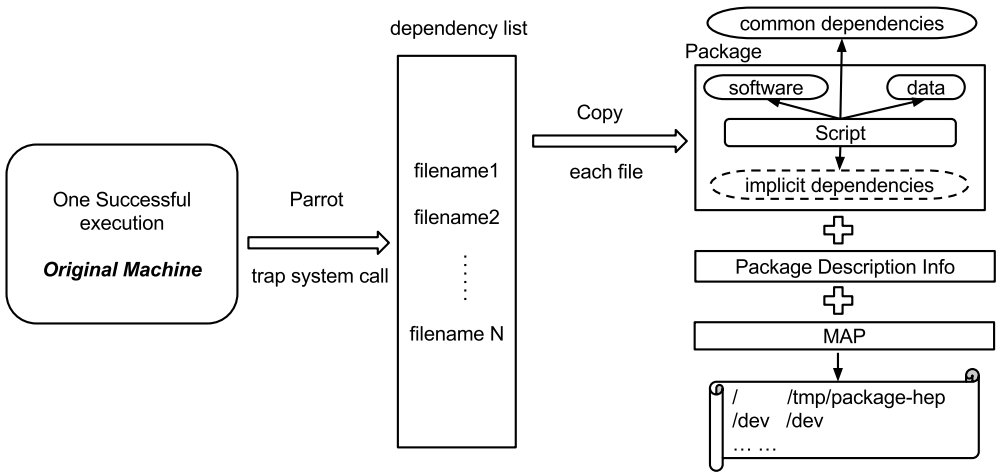

Figure 4: Workflow of The Fine-Grained Dependency Management Toolkit Based on Parrot

	Shallow Copy	Medium Copy
Whole Files:	1632	15642
Empty Files:	14273	263
Directories:	1549	1549
Symbolic Links:	4614	4614
Total Size:	**21GB**	**28GB**

Table 3: Package Information

Task Category	Original Script	Reduced Package
Obtain Namelist	N/A	28min 28s
Generate Package	N/A	85min 51s
Software Acquisition	8min 11s	N/A
Environment Build	5min 49s	4s
Analysis Code	20min 31s	13min 04s

Table 4: Execution Time Comparison between Script-Solution and Package-Solution

We evaluated the correctness of the reduced package and the overhead of generating the package. To do it, the application was repeated with the help of Script (as shown in Figure 2) on the original machine, mentioned as **Script-Solution** in the following part, one reduced package was generated and verified with the help of Parrot on the original machine, which will be mentioned as **Package-Solution**. Then, two different virtual machines (VM) - one VM from Notre Dame Cloud Platform based on KVM and one VM from Amazon EC2 Platform based on Xen, were employed to further verify the correctness of the reduced package.

We first repeated the example from scratch using the Script translated from the original email on the original machine and counted the time consumption and data size. Then based on the successful execution of Script-Solution, Package-Solution is evaluated - one package containing necessary dependencies is generated, and then the time consumption and data size is analyzed.

To measure time consumption, we counted the time used to obtain remote software dependencies, build environment, and analyze the dataset respectively. We also counted the time used to obtain the filename list and generate the package. To measure data size, we can easily figure out the size of each data and code source inside the package under Package-Solution. Script-Solution does not support mining of implicit dependencies. As for remote sources, we can figure out the named size through the analysis script. However, it is hard to figure out the named size of each local source. Instead, we only knew that total size of each file system is very large.

Table 4 shows the execution time comparison between Script-Solution and Package-Solution. Package-Solution is faster than Script-Solution, because all the software copied into the package has been compiled and the Software Acquisition stage is not necessary. and all the environment building only takes 4 seconds. We were surprised that Package-Solution even reduces the actual analysis time. The reason for this is that data is obtained through accessing HDFS in Script-Solution, but is copied into the package in Package-Solution. This localization of experimental data speeds up the data analysis process, resulting the actual analysis time reducing from 20 minutes to 13 minutes.

Table 4 also illustrates the time used to obtain the filename list and generate the package. The time used for these two steps is longer than the execution time, because each filename of the list, together with its system call type, needs to be checked, and the structure of each directory item must be maintained. However, this is only done once. Once the package is generated, many users can directly obtain the package and repeat the application separately.

Table 1 illustrates the total size, named size in the example and actually used size of each remote source (the first 4 items) and local source (the remaining 5 items). The third column corresponds to the data size of Package-Solution and can be easily figured out, because all the necessary data has been copied into the package. Script-Solution does not support measuring implicit dependencies. As for remote sources, we can figure out the named size through the analysis script. However, it is hard to figure out the named size of each local source. Instead, we only knew that total size of each file system is very large.

To further verify the correctness of Package-Solution on

Machine Type	Distro Version	CPU Cores	Mem (GB)	Execution Time
Native Machine	Red Hat 5.10	64	125	13min 04s
KVM (Notre Dame)	CentOS 5.10	4	2	21min 38s
Xen (EC2)	Red Hat 5.9	16	60.5	13min 30s

Table 5: Evaluation of Different Machines

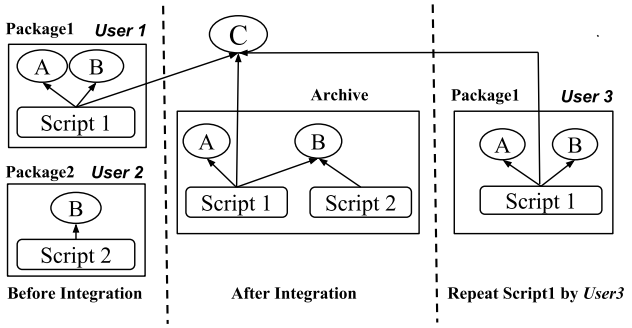

Figure 5: Preserving Multiple Artifacts

other machines, two different machines are employed - one virtual machine [14] from Notre Dame Cloud Platform based on KVM sharing the same kernel version with the original machine, and one virtual machine from amazon EC2 [2] based on Xen.

Table 5 illustrates the configuration of each machine and the execution time of the application on each machine. All the machines adopt x86_64 hardware platform and Linux OS. Both of the two VMs repeated the application with the help of the package generated on the original machine successfully. The execution time on one machine greatly depends on its hardware configuration.

Regardless, we demonstrate that the application runs correctly in a completely different environment.

7. OPEN PROBLEMS

Figure 5 shows the rough information architecture of the archive that we imagine for complex scientific software like TauRoast. Each artifact to be preserved is a package that consists of a top-level script to invoke the software, a dependency map, and the dependencies themselves, which may be external to the original program. The artifacts are then ingested into the archive, where shared dependencies are stored only once. In cases where an artifact has a dependency on a trusted (or very large) remote archive, the dependency may simply be tracked instead of ingested. A researcher that wishes to reproduce a given result need only refer to the unique identifier of the artifact, and will be able to automatically extract all of the dependent components of that artifact.

For example, in Figure 5 Script 1 depends on items A, B, and C. Items A and B are ingested into the archive, where B is shared with Script 2. Item C is stored in another trustworthy repository and is tracked rather than ingested. When Script 1 is exported from the repository, items A and B are exported along with it, while C can be copied or remain remote, according to the end user's choice.

As simple as that picture appears, there are a number of problems that must be solved to get there:

Measure the Mess or Force Cleanliness? Two radically different approaches to dependency tracking are possible. The first is to allow end users complete freedom to construct their environment as desired, then *measure* what items were actually used. As we have shown, this is possible, but has significant overheads and does not fully preserve the structure or intent of the end user. The second approach is to *force* users to work in a clean environment in which no resource can be used until a proper dependency has been declared. This ensures that all dependencies are known in advance (and made explicit to the end user) but places a variety of restrictions on the user's daily work, and may prevent creative approaches that do not fit within the curator's view of how programs should be structured. Whether end users will accept the inconvenience of forced cleanliness for the benefits of reproducibility can only be discovered through experiment.

Granularity of Dependencies. Dependencies could be handled at many different levels of granularity. In this work, we have shown how they can be handled at the level of entire repositories or individual files. Other possible choices might include intermediate-sized software packages (like RPM) or in the case of experimental data, even portions of individual large files. Clearly, a larger granularity will result in fewer packages, simpler dependency maps, and more wasted space; smaller granularity results in complex dependency maps and less wasted space. A hybrid solution may be able to combine both by storing large granularity dependencies, but retain the ability to select sub-items out of objects when efficiency demands it.

Scope of Reuse. We have presented the data preservation problem as primarily one of accurate reproduction: if a result depends on running program X, we must be able to run exactly X again. However, the goal of scientific reproducibility is rarely limited to running *precisely* what a predecessor did. Often, the objective is to change a parameter or a data input in order to see how the result is affected. To that end, the preservation system must capture enough of the surrounding material to permit modifications to succeed. From this perspective, a larger granularity of preservation is desirable. A better understanding of how end users will consume preserved software will help to shape how software is preserved in the first place.

Dependency Detection. If we allow users to work in uncontrolled environments, then we must have better methods for understanding the dependencies of existing programs. At first, we relied on an expert reader to examine the user's script and extract the repository dependencies. This is clearly not a scalable approach. We then demonstrated an automated method of observing what individual files are accessed by a program. However, this does not cover all types of dependencies, particularly those that are networked. More sophisticated observation techniques could infer higher-level information, such as the RPM package names of the files accessed, the URLs of remote repositories named throughout the program, or even the addresses and names of networked dependencies like databases and filesystems.

Source, Binary, or Both? A science archive might choose to retain the source code of an artifact or the bi-

nary code that can actually run in a given environment. While conventional wisdom suggests that access to source code is critical for the long term survival and evolution of a piece of software, it also requires the maintenance of an enormous amount of supporting software in the form of compilers, linkers, and supporting libraries for the target platform. Rebuilding all of these for every invocation of an artifact is likely to have excessive cost. We suggest that a realistic repository will have to maintain *both*: the source describes the ultimate meaning of the code, but the binary is an important performance cache, a backup if the compiler toolchain should fail to be preserved, and a checksum to ensure that a source artifact was rebuilt correctly.

8. RELATED WORK

Generally, there are three approaches to preserve software environment: hardware preservation, migration and emulation. Hardware preservation preserves the original software and its original operating environment. Software migration technique [9, 21] was used to facilitate running software on new machines. However, migration often involves the recompiling and re-configuring the source code to accustom a new hardware platform and software environment. Emulation recreates the original software and hardware environment by programming future platforms and OSs. One common solution to implement this is virtual machine. According to the usage and emulation degree of the real machine, virtual machine can be divided into system virtual machine and process virtual machine. The working principle, design principle and performance evaluation of system virtual machine were illustrated in [14, 30]. The functionality of system VM to support different guest operating systems was illustrated in [3, 19, 27]. F. Esquembre [13] illustrated how JVM, one process virtual machine, can expedite the creation of scientific simulations in Java. The pros and cons of these three approaches were discussed in [24, 25, 16].

The preservation of computing environment and software environment was treated as one entirety in [24, 25, 16]. However, frequently changing experiment software makes the maintenance of the preserved experimental environment very complex. CernVM [7] treated them as two different categories. The preservation of computing environment is implemented with CernVM, and the preservation of software environment is based on a CernVM filesystem(CVMFS) specifically designed for efficient software distribution.

The importance of preserving software in source code format was emphasized in [34, 8]. However, CVMFS [7] published pre-built and configured experiment software releases to avoid repeating the time-consuming software building procedure.

Attempts from different perspectives to facilitate the reproduction of scientific experiments utilizing preserved software library has been made. The software distribution mechanism over network was discussed in [12, 4]. J. R. Rice et al. [26] made the reproduction process easier through the integration of user interface, scientific software libraries, knowledge base into problem-solving environment. S. R. Kohn et al. [20] tried to enable the creation and distribution of language-independent software library by addressing language interoperability. a scalable, distributed and dynamic workflow system for digitization processes was proposed in [29]. A distributed archival network was designed in [31] to facilitate process-oriented automatic long-term digital preservation. M. Agosti et al. [1] aimed to help non-domain users to utilize the digital archive system developed for domain experts.

Current mechanisms of preserving scientific experiments assume that all the data and software mentioned in the experiments are necessary for the reproduction of the experiments. However, this is not always right. In some cases, the original author may leave additional code referring to irrelative data and software in the experiment programs. One mechanism, which can figure out the absolutely relevant data and software of one experiment, is important for both the preservation and reproduction of scientific experiments.

B. Matthews et al. [22] introduced one conceptual framework for software preservation from several case studies of software preservation. One tool to capture software preservation properties within a software environment was designed in [23] through a series of case studies conducted to evaluate the software preservation framework. L. R. Johnston et al. [18] proposed one overall data curation workflow for 3-5 case studies of preserving research data. Two case studies [5] were conducted to figure out the properties of data to be reused in the future, including type, purpose, new users. To figure out how to preserve HEP applications, this paper studies one case of preserving one representative HEP application.

9. POSTSCRIPT

We began this work in late 2013, generating a reduced package for *TauRoast* based on the configuration of a standard machine at Notre Dame at the time. In the course of writing this paper in spring 2014, almost everything about the computing environment changed: the operating system was upgraded, a new version of CMSSW was released, and our local HEP users switched from using CVS to CVMFS for accessing CMSSW. The original script provided by the author failed to run on the same machine. But, the reduced package we created three months earlier in the old environment worked just fine on the new machine.

Acknowledgments

This work was supported in part by National Science Foundation grants PHY-1247316 (DASPOS) and OCI-1148330 (SI2). We also thank...

10. REFERENCES

[1] M. Agosti and N. Orio. To envisage and design the transition from a digital archive system developed for domain experts to one for non-domain users. In *Proceedings of the 12th ACM/IEEE-CS joint conference on Digital Libraries*, pages 11–14. ACM, 2012.

[2] E. Amazon. Amazon elastic compute cloud (Amazon EC2). *Amazon Elastic Compute Cloud (Amazon EC2)*, 2010.

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. *ACM SIGOPS Operating Systems Review*, 37(5):164–177, 2003.

[4] J. Blomer, P. Buncic, and T. Fuhrmann. CernVM-FS: delivering scientific software to globally distributed

computing resources. In *Proceedings of the first international workshop on Network-aware data management*, pages 49–56. ACM, 2011.

[5] C. L. Borgman. Data, data use, and scientific inquiry: Two case studies of data practices. In *Proceedings of the 12th ACM/IEEE-CS joint conference on Digital Libraries*, pages 19–22, 2012.

[6] D. Borthakur. HDFS architecture guide. *HADOOP APACHE PROJECT http://hadoop. apache. org/common/docs/current/hdfs design. pdf*, 2008.

[7] P. Buncic, C. A. Sanchez, J. Blomer, L. Franco, A. Harutyunian, P. Mato, and Y. Yao. CernVM–a virtual software appliance for LHC applications. In *Journal of Physics: Conference Series*, volume 219, page 042003. IOP Publishing, 2010.

[8] M. Castagné. Consider the Source: The Value of Source Code to Digital Preservation Strategies. *SLIS Student Research Journal*, 2(2):5, 2013.

[9] C. Cifuentes and V. Malhotra. Binary translation: Static, dynamic, retargetable? In *Software Maintenance 1996, Proceedings., International Conference on*, pages 340–349. IEEE, 1996.

[10] C. Collaboration, S. Chatrchyan, et al. The CMS experiment at the CERN LHC. *Jinst*, 3(08):S08004, 2008.

[11] C. Collaboration et al. The CMSSW Application Framework, 2006.

[12] G. Compostella, S. P. Griso, D. Lucchesi, I. Sfiligoi, and D. Thain. CDF software distribution on the Grid using Parrot. In *Journal of Physics: Conference Series*, volume 219, page 062009. IOP Publishing, 2010.

[13] F. Esquembre. Easy Java Simulations: A software tool to create scientific simulations in Java. *Computer Physics Communications*, 156(2):199–204, 2004.

[14] R. P. Goldberg. Survey of virtual machine research. *Computer*, 7(6):34–45, 1974.

[15] K. Holtman. CMS data grid system overview and requirements. Technical report, CERN-CMS-NOTE-2001-037, 2001.

[16] N. C. Hong, S. Crouch, S. Hettrick, T. Parkinson, and M. Shreeve. Software Preservation Benefits Framework. *Software Sustainability Institute Technical Report*, 2010.

[17] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems (TOCS)*, 6(1):51–81, 1988.

[18] L. R. Johnston. A Workflow Model for Curating Research Data in the University of Minnesota Libraries: Report from the 2013 Data Curation Pilot. 2014.

[19] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. kvm: the Linux virtual machine monitor. In *Proceedings of the Linux Symposium*, volume 1, pages 225–230, 2007.

[20] S. R. Kohn, G. Kumfert, J. F. Painter, and C. J. Ribbens. Divorcing Language Dependencies from a Scientific Software Library. In *PPSC*, 2001.

[21] D. Mancl. Refactoring for software migration. *Communications Magazine, IEEE*, 39(10):88–93, 2001.

[22] B. Matthews, B. McIlwrath, D. Giaretta, and E. Conway. The significant properties of software: A study. *JISC report, March*, 2008.

[23] B. Matthews, A. Shaon, J. Bicarregui, and C. Jones. A framework for software preservation. *International Journal of Digital Curation*, 5(1):91–105, 2010.

[24] B. Matthews, A. Shaon, J. Bicarregui, C. Jones, J. Woodcock, and E. Conway. Towards a methodology for software preservation. 2009.

[25] T. A. Phelps and P. B. Watry. A no-compromises architecture for digital document preservation. In *Research and Advanced Technology for Digital Libraries*, pages 266–277. Springer, 2005.

[26] J. R. Rice and R. F. Boisvert. From scientific software libraries to problem-solving environments. *Computational Science & Engineering, IEEE*, 3(3):44–53, 1996.

[27] M. Rosenblum. VMware's virtual platform. In *Proceedings of hot chips*, pages 185–196, 1999.

[28] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun network filesystem. In *Proceedings of the Summer USENIX conference*, pages 119–130, 1985.

[29] H. Schöneberg, H.-G. Schmidt, and W. Höhn. A scalable, distributed and dynamic workflow system for digitization processes. In *Proceedings of the 13th ACM/IEEE-CS joint conference on Digital libraries*, pages 359–362. ACM, 2013.

[30] J. E. Smith and R. Nair. The architecture of virtual machines. *Computer*, 38(5):32–38, 2005.

[31] I. Subotic, L. Rosenthaler, and H. Schuldt. A distributed archival network for process-oriented autonomic long-term digital preservation. In *Proceedings of the 13th ACM/IEEE-CS joint conference on Digital libraries*, pages 29–38. ACM, 2013.

[32] D. Thain and M. Livny. Parrot: An application environment for data-intensive computing. *Scalable Computing: Practice and Experience*, 6(3):9–18, 2005.

[33] B. Welch, M. Unangst, Z. Abbasi, G. A. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou. Scalable Performance of the Panasas Parallel File System. In *FAST*, volume 8, pages 1–17, 2008.

[34] J. G. Zabolitzky. Preserving software: Why and how. *Iterations: An Interdisciplinary Journal of Software History*, 1(13):1–8, 2002.